UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Búsqueda en rejilla no regular

Autor: David López Ramos
Tutor: Carlos María Alaíz Gudín
Ponente: José Ramón Dorronsoro Ibero

junio 2019

David López Ramos
Búsqueda en rejilla no regular

David López Ramos
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

¿Un número primo concreto?
De acuerdo, tomemos el 57.

Alexandre Grothendieck

Agradecimientos

A mi familia, por ser un apoyo constante y darme la oportunidad que ellos no tuvieron.

A mis compañeros, por las experiencias y conocimientos compartidos en estos fantásticos años.

A mi profesora de matemáticas del instituto, por empujarme a estar aquí.

Resumen

El *machine learning* o aprendizaje automático es una rama de la inteligencia artificial (IA) que proporciona a los sistemas la capacidad de aprender y mejorar automáticamente a partir de la experiencia, sin ser programados de manera explícita. El objetivo principal es reducir el error cometido por estos sistemas, optimizando (minimizando) la función que representa dicho error.

Los métodos de optimización usados antes de la aparición de los ordenadores de alta velocidad eran indirectos, se utilizaban las propiedades de la función (derivadas y condiciones de optimalidad). Sin embargo, con el avance de la capacidad de cómputo, se han extendido los métodos directos, en los que la función representa una caja negra y se van realizando sucesivas pruebas observando los valores de entrada y salida de la función, sin usar herramientas analíticas de la misma. Éstos tienen la ventaja de ser muy sencillos de entender, aunque son menos eficientes que los indirectos.

Este Trabajo de Fin de Grado se centra en el desarrollo de una modificación del método clásico de búsqueda en rejilla regular, perteneciente a los métodos directos de optimización, pues el proceso original presenta limitaciones y es muy ineficiente a nivel de computación. Para dicha tarea se realizarán variaciones partiendo de la rejilla regular, de forma que los puntos queden distribuidos en el espacio de una forma más estratégica para la posterior evaluación de los hiperparámetros. También se creará una variante de la búsqueda aleatoria para evaluar su rendimiento.

Este documento presenta las diferentes fases llevadas a cabo en el diseño inicial, la codificación de la rejillas modificadas, y las posteriores pruebas de rendimiento de las mismas, comparándolas con las originales. Al final, como conclusión, observaremos que nuestras nuevas rejillas no regular y aleatoria modificada se comportarán bien en la minimización de funciones aleatorias. Respecto a la optimización de hiperparámetros en un modelo de aprendizaje automático, las nuevas rejillas tendrán un rendimiento mejor o igual que la regular en nuestros experimentos, superándola claramente en dos de los cuatro experimentos realizados.

Palabras clave

Aprendizaje automático, Optimización, Hiperparámetros, Búsqueda en rejilla

Abstract

The machine learning is a branch of artificial intelligence (AI) that provides systems the ability to learn and automatically improve from the experience without being explicitly programmed. The main goal is to reduce the error of the systems, optimizing (minimizing) the function that represents that error.

The optimization methods used before the appearance of high-speed computers were indirect, the properties of the function (derivate and optimality conditions) were used. However, with the advances in computational capacity, direct methods have extended. In these methods, the function is considered a black box and successive tests are performed looking at the input and output values of the function, without using analytical tools. They have the advantage of being very simple to understand, although they are less efficient than indirect methods.

This Final Degree Project aims to develop a modification of the classical method of regular grid search, belonging to direct optimization methods, because the original method has limitations and it is computationally very inefficient. In particular, variations will be made starting from the regular grid, so that the points are distributed in the space in a more strategic way for the subsequent evaluation of the hyperparameters. In addition, a variant of the random search will be created to evaluate its performance.

This document presents the different phases carried out in the initial design, the implementation of the modified grids, and the subsequent numerical experiments, comparing them with the original ones. At the end, as a conclusion, we will observe that our new grids, the not regular and the modified random ones will behave well in the minimization of random functions. In the optimization of hyperparameters in an machine learning model, the new grids will have a better or equal performance than the regular one in our experiments, improving it clearly in two of the four experiments done.

Keywords

Machine Learning, Optimization, Hyperparameter, Grid search

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de figuras

Lista de tablas

1

Introducción

1.1. Motivación

Este proyecto pretende aprovechar la potencia computacional moderna para trabajar con los métodos de optimización directos, en los que vemos la función como una caja negra sobre la que vamos realizando sucesivas entradas y observando los valores de salida.

En especial, nos centraremos en el método de búsqueda en rejilla regular, el que tomaremos como base para realizar una serie de modificaciones que puedan mejorar su rendimiento en el ámbito de la optimización.

1.2. Objetivos

El objetivo de este trabajo es la implementación de un método de optimización tomando como referencia la búsqueda en rejilla regular, el cual puede suponer una mejora en la búsqueda de hiperparámetros en el aprendizaje automático.

Para ello, se estudiará previamente la distribución de los puntos del método original. Después, se propondrá una solución para una redistribución más óptima de esos puntos, de forma que, finalmente, se puedan poner a prueba las modificaciones y constatar o no la mejora en el entrenamiento y predicción del modelo.

Como lenguaje de desarrollo se utilizará Python, por contar con una serie de librerías centradas en el Machine Learning, como es Sklearn [1]. Los métodos aquí proporcionados serán de gran ayuda a la hora de realizar las pruebas de rendimiento de la solución propuesta. Además, Python es un lenguaje flexible y rápido, en el que la codificación resulta muy simplificada.

1.3. Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1**: en él se realiza una breve introducción del proyecto, explicando el objetivo del mismo, realizar unas modificaciones del método clásico de rejilla regular como posible mejora para la búsqueda de hiperparámetros.
- **Capítulo 2**: aquí se definen los conceptos iniciales sobre el contexto que se va a tratar, tales como aprendizaje automático, hiperparámetros y validación cruzada. Además, se hace un repaso del panorama actual en la búsqueda de hiperparámetros, así como una comparación, basada en un artículo, entre el rendimiento en el aprendizaje automático de la rejilla regular y la rejilla aleatoria.
- **Capítulo 3**: en esta parte, analizamos el diseño de la actual rejilla regular y proponemos nuevos algoritmos cuyas proyecciones de puntos son equidistantes, rejilla no regular y rejilla aleatoria equidistante, para ver su posterior desempeño en una serie de pruebas.
- **Capítulo 4**: la parte de experimentos. Aquí compararemos el rendimiento de nuestras nuevas rejillas frente a la rejilla regular y la rejilla aleatoria. Realizaremos un análisis sobre los distancias entre puntos de cada una de ellas, así como una prueba de minimización de funciones de la que elaboraremos un *ranking* de posiciones. Por último, utilizaremos las rejillas para la optimización de hiperparámetros mediante la librería Sklearn de Python.
- **Capítulo 5**: aquí se exponen las conclusiones obtenidas de los experimentos anteriores, comparando el rendimiento conseguido por cada una de las rejillas. Posteriormente, se detallan una serie de tareas que podrían realizarse en un futuro para evaluar cómo se comportan los nuevos métodos aportados en un marco que podría ser más favorable.

2

Aprendizaje automático y búsqueda de hiperparámetros

En este capítulo hablaremos sobre el panorama actual en la búsqueda de hiperparámetros. Para ello, empezaremos definiendo qué es el aprendizaje automático y la importancia de dichos hiperparámetros a la hora de entrenar un modelo. Posteriormente, comentaremos la validación cruzada, un método para tratar de estimar de forma estable el rendimiento en predicción de nuestro modelo.

Después, compararemos los métodos de búsqueda en rejilla regular frente al método de búsqueda aleatoria, basándonos en el artículo *Random Search for Hyper-Parameter Optimization*, de James Bergstra y Yoshua Bengio [2] .

2.1. Hiperparámetros en el aprendizaje automático

¿Qué es el aprendizaje automático?

El aprendizaje automático es una rama de la inteligencia artificial (IA) que proporciona a los sistemas la capacidad de aprender y mejorar automáticamente a partir de la experiencia sin ser programados de manera explícita [3] [4]. Es decir, los sistemas utilizan datos que les proporcionamos para aprender por sí mismos y ser capaces de reconocer patrones y tomar decisiones en base a ello para los nuevos ejemplos de datos.

Existen varios tipos de aprendizaje automático, dependiendo de las necesidades del problema:

Aprendizaje supervisado: se genera un modelo predictivo en base a un conjunto de pares de entradas y salidas (conocimiento *a priori*) con el que se realiza el ajuste del modelo inicial. Así, el algoritmo aprende a estimar la salida para las nuevas muestras, pudiendo comprobar su nivel de acierto. La salida a predecir puede ser cuantitativa (como en los problemas de regresión) o cualitativa (como en los problemas de clasificación).

Aprendizaje no supervisado: similares a los anteriores, con la diferencia de que éstos ajustan el modelo predictivo teniendo en cuenta solo los datos de entrada, sin que haya una salida definida. Por tanto, los datos de entrada no están etiquetados, pues no se necesita

para entrenar el modelo.

Aprendizaje por refuerzo: se interáctua con el entorno mediante acciones y descubriendo errores o recompensas. Las características más relevantes son la búsqueda por prueba y error y la recompensa obtenida. Se necesita un simple *feedback* de recompensa para que el agente sepa qué acción es la mejor.

Parámetros estándar e hiperparámetros

En el aprendizaje automático contamos con dos tipos diferentes de parámetros: los estándar y los que denominamos hiperparámetros [5].

Un modelo de aprendizaje automático es, a grandes rasgos, una fórmula matemática con una serie de parámetros que pueden ser determinados y aprendidos (es decir, ajustados) a partir de los datos. Por tanto, el objetivo es ajustar un modelo a los datos variando los parámetros, a través del entrenamiento del modelo.

Por otra parte, los hiperparámetros son aquellos que no pueden ser aprendidos directamente de los datos, pues expresan propiedades de un nivel superior en el modelo, tales como la complejidad o la rapidez de entrenamiento del mismo. Los hiperparámetros se suelen predefinir antes de comenzar el entrenamiento en base a pruebas con diferentes valores en diferentes modelos, observando cuáles funcionan mejor. Es decir, vemos la función como una caja negra, en la que insertamos distintas configuraciones de hiperparámetros y observamos la salida para comprobar qué configuración ha obtenido un mejor rendimiento. Normalmente el proceso de exploración de hiperparámetros de forma manual es muy laborioso y el espacio de búsqueda muy extenso, provocando que evaluar cada configuración sea una tarea costosa.

Ejemplos de hiperparámetros en modelos reales pueden ser: tasa de aprendizaje, número de capas ocultas en una red neuronal, número de hojas o profundidad en un árbol...

2.2. Validación cruzada

Es necesario estimar el rendimiento que tendría un modelo de aprendizaje automático cuando haga predicciones sobre nuevos ejemplos. Esto es, hay que tener garantías de que el ajuste realizado a los datos de entrenamiento tendrá buen funcionamiento cuando se use con los nuevos datos, aquellos que no se han visto aún. Por tanto, surge la necesidad de tener algún método que compruebe que hemos obtenido un buen ajuste a los datos y que no hay demasiado ruido, o en otras palabras, que no esté muy sesgado. Aquí entra en juego la validación cruzada [6].

La validación cruzada es un proceso de remuestreo que se utiliza para evaluar modelos en el aprendizaje automático si se cuenta con una muestra de datos limitada. Tiene un parámetro k que es

el número de hojas, es decir, el número de grupos en los que se dividirá la muestra de datos. Este procedimiento se usa para estimar la eficiencia de un modelo de aprendizaje automático con nuevos datos. En resumen, con las muestras de datos disponibles se hacen estimaciones sobre el desempeño que tendrá el modelo con las nuevas muestras que no se han usado en el entrenamiento.

Es un método muy popular por su facilidad de entendimiento, y porque proporciona una estimación menos sesgada que otros métodos que usan una simple división de los datos en conjunto de entrenamiento y prueba. El modo de actuar es el siguiente:

1.– Se mezcla el conjunto de datos al azar.

2.– Se divide el conjunto en k grupos.

3.– Para cada grupo:

  3.1.– Se coge el grupo como conjunto de prueba.

  3.2.– Se cogen los demás grupos como un conjunto de entrenamiento.

  3.3.– Se ajusta el modelo con el conjunto de entrenamiento.

  3.4.– Se evalúa el modelo con el conjunto de prueba.

  3.5.– Se guarda la puntuación obtenida al evaluar y se descarta el modelo.

4.– Se calcula la eficiencia del modelo promediando todas las puntuaciones obtenidas.

Por tanto, cada grupo se utiliza como conjunto de prueba una vez y como conjunto de entrenamiento $k - 1$ veces.

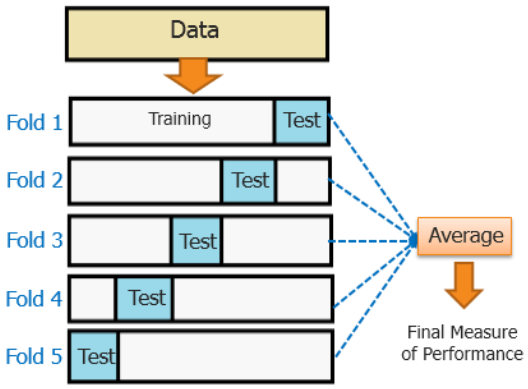


Figura 2.1: Procedimiento de validación cruzada de 5 hojas [7].

2.3. Búsqueda en rejilla regular vs búsqueda aleatoria

En el artículo mencionado anteriormente [2] se realizan una serie de pruebas empíricas y teóricas para determinar los hiperparámetros, mediante dos de las técnicas más utilizadas hoy en día:

búsqueda en rejilla regular y búsqueda aleatoria.

Cabe destacar que existen varias razones por las cuales la búsqueda manual y la búsqueda en rejilla se siguen usando actualmente a pesar de las décadas de investigación en optimización global:

- La optimización manual proporciona a los investigadores cierto grado de comprensión de la función.
- No hay límites técnicos para la optimización manual.
- La búsqueda en rejilla es fácil de implementar y la paralelización es trivial.
- La búsqueda en rejilla generalmente encuentra mejores hiperparámetros que la puramente manual en el mismo tiempo.
- La búsqueda en rejilla es efectiva en espacios de dimensiones bajas (por ejemplo, 1-d, 2-d).

En [2] se muestra que la búsqueda aleatoria tiene todas las ventajas prácticas de la búsqueda en rejilla (conceptualmente simple, fácil de implementar y paralelizar) y además proporciona una ligera reducción de eficiencia en dimensiones bajas para obtener un gran aumento de eficiencia en dimensiones elevadas. Esto se debe a que la función objetivo suele tener algunas dimensiones poco efectivas, es decir, la función es más sensible a cambios en unas dimensiones que en otras. Por ejemplo, en dos dimensiones esto se expresaría así: si una función f de dos variables puede ser aproximada de forma razonable por otra función de una variable, $f(x, y) \approx g(x)$, podemos decir que f tiene una dimensión poco efectiva.

Se realizó un estudió en redes neuronales, configurándolas con ambos métodos. El resultado fue que la búsqueda aleatoria sobre el mismo dominio conseguía encontrar modelos tan buenos o incluso mejores que la búsqueda en una rejilla regular, especialmente cuando se trataba de dimensiones elevadas (más de 2) pues ahí la mejora de rendimiento de la aleatoriedad era más notable, otorgando la misma capacidad computacional a ambas, es decir, evaluando el mismo número de puntos.

Un análisis gaussiano de la función de los hiperparámetros determinó que los resultados son debidos a que solo son relevantes algunos de esos hiperparámetros para los conjuntos de datos utilizados. Es lo que hemos comentado sobre la dimensión poco efectiva de la función, lo que provoca que la búsqueda regular evalúe puntos en dimensiones que tienen muy poca relevancia, obteniendo un peor rendimiento.

Podríamos pensar que si conociéramos, de antemano, qué subespacios son los importantes, podríamos diseñar una rejilla apropiada para la búsqueda. Sin embargo, según el artículo, esto no sería de ayuda, ya que en diferentes conjuntos de datos, también son diferentes los subespacios que son importantes y en diferentes grados. Una rejilla con suficiente granularidad para optimizar los hiperparámetros de todos los conjuntos de datos sería, consecuentemente, ineficiente para cada conjunto de datos individualmente. Por otro lado, la búsqueda aleatoria funciona bien con la dimensión poco efectiva, puesto que tiene la misma eficiencia en los espacios importantes que si se hubiera usado para

buscar sólo en estas dimensiones importantes.

Los experimentos aleatorios también son más fáciles de realizar que los experimentos de rejilla por razones relacionadas con la independencia estadística de cada prueba:

- El experimento se puede detener en cualquier momento y las pruebas forman un experimento completo.
- Si hay computadoras adicionales disponibles, se pueden agregar nuevas pruebas a un experimento sin tener que ajustar la rejilla.
- Cada prueba se puede realizar de forma asíncrona.
- Si la computadora que realiza una prueba falla por algún motivo, su prueba puede ser abandonada o reiniciada sin poner en peligro el experimento.

Por tanto, tras lo expuesto anteriormente, los autores del artículo recomiendan utilizar la búsqueda aleatoria en lugar de la búsqueda en rejilla a la hora de optimizar hiperparámetros.

3

Diseño y desarrollo de las nuevas rejillas

Este capítulo está dedicado al diseño y las decisiones de implementación de las rejillas utilizadas para la posterior optimización, especialmente, los nuevos métodos no regular y aleatorio equidistante.

3.1. Diseño de las rejillas para la optimización

3.1.1. Rejilla regular

La búsqueda en rejilla regular prueba todas las posibles combinaciones de todas las dimensiones, es decir, el producto cartesiano de los valores.

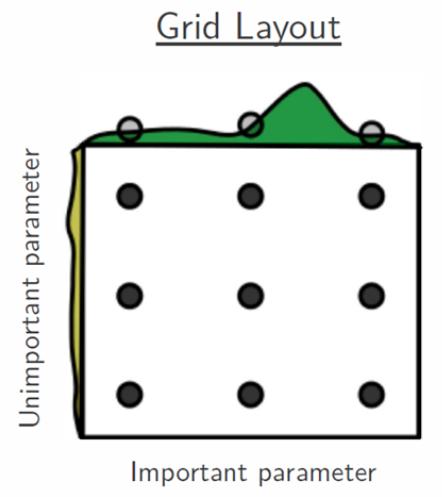


Figura 3.1: Rejilla regular 3x3 [2].

Por ejemplo, en el caso bidimensional, teniendo 3 valores en cada dimensión, con 9 puntos en total, obtenemos la figura 3.1. En ella observamos que cada cada valor del eje X se prueba 3 veces (pues es el número de valores del eje Y). Por tanto, podemos ver fácilmente cómo afecta en este método la dimensión poco efectiva de la función: si una dimensión no fuera importante, tan solo tendríamos 3 puntos (pertenecientes a la otra dimensión) para evaluar la función, debido a la distribución de los

```
input  : vector numbers de puntos por dimensión
output: rejilla regular de dimensión Len (numbers)
for n in numbers do
    añadimos los distintos puntos que tendrá cada variable, en [0,1];
    variables ← variables + Linspace( 0, 1, n );
end
vectores ← Meshgrid( variables );
```

Algoritmo 3.1: Pseudocódigo de creación de la rejilla regular.

puntos de forma regular. Veamos el algoritmo que hemos utilizado para crear esta rejilla.

Como se puede apreciar en el algoritmo 3.1, construiremos nuestra rejilla inicialmente en el intervalo $[0, 1]$ pues es fácilmente reescalable a otros intervalos, como comprobaremos próximamente en los experimentos.

Nuestra misión, por tanto, es buscar una nueva distribución de los puntos de la rejilla para que la dimensión poco efectiva no afecte a la otra. Para ello, partiremos de la rejilla regular y deberemos ir desplazando los puntos una cierta cantidad, de forma que no se repitan valores ni en las filas ni en las columnas.

3.1.2. Rejilla no regular

Idea: hipercubo latino

Algo parecido a lo que queremos ocurre en el hipercubo latino [8]. En él, los puntos se sitúan aleatoriamente dentro de celdas diferentes de forma que ninguna fila ni columna se seleccione más de una vez. Para los jugadores de ajedrez, algo similar sucede en el famoso problema de las ocho reinas en el tablero, que consiste en colocar en él ocho reinas sin que se ataquen entre ellas. A su vez, los conocidos sudokus también trabajan con este problema, ya que consiste en el reparto de objetos (en este caso, números) bajo ciertas restricciones.

Por tanto, la forma de operar para conseguir un hipercubo latino consiste en seleccionar una celda al azar, generar un punto en esa celda, descartar su fila y su columna y repetir hasta tener todos los puntos. Más tarde, definiremos el método aleatorio equidistante, el cual también tendrá una componente de aleatoriedad, pero se basará en permutaciones de vectores con puntos equidistantes ya fijados en cada eje.

En la figura 3.2 se puede ver un ejemplo del hipercubo latino en 2D.

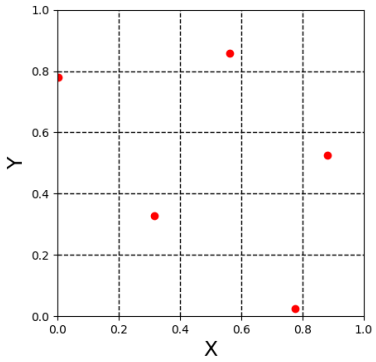


Figura 3.2: Ejemplo de hipercubo latino en dos dimensiones [9].

Construcción de la nueva rejilla no regular

Visto el diseño de la rejilla regular, queremos definir una nueva rejilla de manera que las proyecciones de los puntos sobre cada eje sean equidistantes. De esta forma, no tendremos proyecciones que se solapan (como ocurre en la regular) y podremos probar una mayor cantidad de valores en cada dimensión.

A continuación, se propone un nuevo algoritmo para crear la rejilla no regular a partir de los vectores de coordenadas de la rejilla regular.

```
input  : Un vector de puntos por coordenada, numbers
output: Vectores coordenadas de la rejilla no regular
creamos los vectores de la rejilla no regular y el vector distancias;
vectores ← RejillaRegular( numbers );
distancias ← CrearDistancias( numbers );
for v ← 0 to Len( vectores ) do
    iteramos sobre los elementos de cada vector;
    for element e in vectores [v] do
        contador ← 0;
        miramos las repeticiones de cada elemento en el vector;
        indices ← Find( e, vectores [v] );
        for element i in indices do
            por cada repetición, desplazamos el punto la distancia correspondiente según el contador;
            vectores [v][i]← vectores [v][i] + contador x distancias [v];
            contador ← contador + 1;
        end
    end
end
```

Algoritmo 3.2: Pseudocódigo de la creación de la rejilla no regular a partir de la rejilla regular.

Sobre el algoritmo anterior, 3.2, la función `RejillaRegular(numbers)`, simplemente devuelve los vectores por coordenadas de la rejilla regular asociada. Respecto a la otra función que aparece, `CreaDistancias(numbers)` devuelve un array con las distancias que debemos desplazar en cada dimensión. Esto es, la distancia en la dimensión i será:

$d_i = \frac{1}{\prod_{j \neq i}^{D} n_j}$, con n_j el número de puntos de la dimensión j y D el número de dimensiones.

La idea es tomar los vectores (valores que tomará cada coordenada) de la rejilla regular e ir desplazando sus puntos repetidos de forma que cubran todo el espacio disponible hasta el siguiente valor. Por ejemplo, partiendo de la rejilla regular 3x3 en el cuadrado $[0, 2]$x$[0, 2]$, tenemos los vectores:

$$x = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \end{bmatrix}$$

$$y = \begin{bmatrix} 0 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 \end{bmatrix}$$

La distancia a desplazar en cada dimensión será $d = \frac{1}{3} = 0{,}33...$. Por tanto, los vectores resultantes, teniendo en cuenta el contador del algoritmo 3.2, serán:

$$x = \begin{bmatrix} 0{,}0 & 0{,}33 & 0{,}66 & 1{,}0 & 1{,}33 & 1{,}66 & 2{,}0 & 2{,}33 & 2{,}66 \end{bmatrix}$$

$$y = \begin{bmatrix} 0{,}0 & 1{,}0 & 2{,}0 & 0{,}33 & 1{,}33 & 2{,}33 & 0{,}66 & 1{,}66 & 2{,}66 \end{bmatrix}$$

Posteriormente, tras su reescalado al cuadrado unidad, quedan:

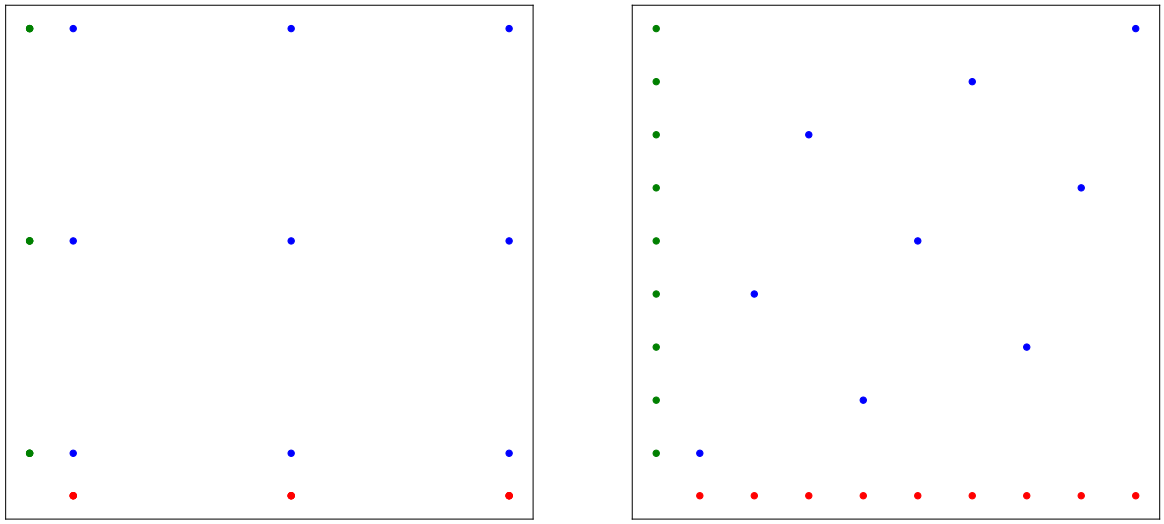
$$x = \begin{bmatrix} 0{,}0 & 0{,}125 & 0{,}25 & 0{,}375 & 0{,}5 & 0{,}625 & 0{,}75 & 0{,}875 & 1{,}0 \end{bmatrix}$$

$$y = \begin{bmatrix} 0{,}0 & 0{,}375 & 0{,}75 & 0{,}125 & 0{,}5 & 0{,}875 & 0{,}25 & 0{,}625 & 1{,}0 \end{bmatrix}$$

Para tener una intuición sobre este proceso de modificación de la rejilla regular, en dimensión 2 podemos pensar que vamos a realizar un *estiramiento*, tirando de los extremos inferior (origen) y superior $(1, 1)$ de forma que no habrá valores repetidos en las filas ni en las columnas, ya que los puntos se habrán desplazado hasta tomar proyecciones equidistantes entre ellos.

Como se aprecia en la figura 3.3(a) tenemos 9 puntos azules dispuestos de forma regular, con 3 valores diferentes en cada una de las dos proyecciones (puntos rojos y verdes). Siguiendo el procedimiento descrito, obtenemos la figura 3.3(b), donde los puntos que antes proporcionaban valores de proyecciones repetidas, ahora ocupan todo el intervalo de proyección $[0, 1]$ obteniendo 9 valores diferentes y equidistantes entre ellos (en cada uno de los ejes).

Es claro ver que gracias a conseguir 9 proyecciones en cada eje y no haber valores repetidos, el problema de la dimensión poco efectiva ya no nos afecta tanto. Si ahora podemos aproximar la función de dos variables $f(x, y)$ a una función de una variable $g(x)$ (de forma razonable) seguiremos teniendo 9 valores diferentes para probar con la dimensión que sí importa, la de x.



(a) Rejilla regular original.

(b) Rejilla no regular, tras desplazar los puntos de la anterior.

Figura 3.3: Modificación de rejilla regular 3x3 para obtener la nueva rejilla. En azul, los puntos de la rejilla, en rojo y verde, sus proyecciones.

3.1.3. Rejilla aleatoria

El método aleatorio, descrito en el algoritmo 3.3, simplemente consiste en tomar puntos al azar en el intervalo indicado. En nuestra implementación, partimos del número total de puntos que queremos (equivalente al producto de puntos por dimensión en la rejilla regular) y creamos ese número de vectores de longitud igual al número de dimensiones elegidas.

input : vector $numbers$ de puntos por dimension
output: rejilla aleatoria de dimensión `Len` $(numbers)$

```
1 definimos las variables utilizadas;
2 totalpoints ← Product ( numbers ) ;
3 dim ← Len ( numbers ) ;
4 obtenemos vectores de puntos aleatorios en [0, 1];
5 vectores ← Random ( totalpoints, dim ) ;
```

Algoritmo 3.3: Pseudocódigo de creación del método aleatorio.

Para crear el método aleatorio se pueden considerar varias distribuciones para obtener puntos aleatorios. En nuestro caso, utilizamos la distribución uniforme en el intervalo $[0, 1]$, en la que todos los puntos tienen la misma probabilidad.

3.1.4. Rejilla aleatoria equidistante

La idea descrita anteriormente en el hipercubo latino la podemos aplicar para la creación de nuestra rejilla aleatoria equidistante, buscando aprovecharnos de la mayor cantidad posible de proyecciones de los puntos. La idea será tomar una rejilla regular muy densa y seleccionar puntos aleatorios sobre ella, pero cuyas proyecciones en los ejes sean equidistantes, teniendo así un cierto control sobre la estructura resultante.

Partiremos, nuevamente, de un array del número de puntos que queremos en cada dimensión (es decir, los parámetros que utilizábamos para modificar la rejilla regular) pero los métodos aleatorios tan solo tendrán en cuenta el número total de puntos, o sea, el producto de puntos de cada dimensión. Con este número total, pongamos N, creamos un vector, pongamos V_0, en el intervalo $[0, 1]$ con N puntos equidistantes. Éste vector son las coordenadas que tomará la primera dimensión. Para hallar el resto de coordenadas, simplemente realizamos permutaciones aleatorias de V_0 y obtenemos las coordenadas del resto de dimensiones, los vectores V_i. El algoritmo 3.4 resume este procedimiento.

input : vector $numbers$ de puntos por dimensión
output: rejilla aleatoria equidistante de dimensión `Len` ($numbers$)

```
definimos las variables utilizadas;
N ← Product( numbers );
dim ← Len( numbers );
V_0 ← Linspace( 0, 1, N );
vectores ← [ V_0 ];
for i ← 1 to dim do
    creamos los distintos vectores permutados por dimensión;
    V_i ← Shuffle( V_0 );
    vectores ← Append( V_i );
end
```

Algoritmo 3.4: Pseudocódigo de creación del método aleatorio equidistante.

Como observamos en la figura 3.4, los puntos están distribuidos aleatoriamente sobre la rejilla (puntos azules) pero sus proyecciones en ambos ejes (puntos rojos y verdes) están situadas de forma equidistante.

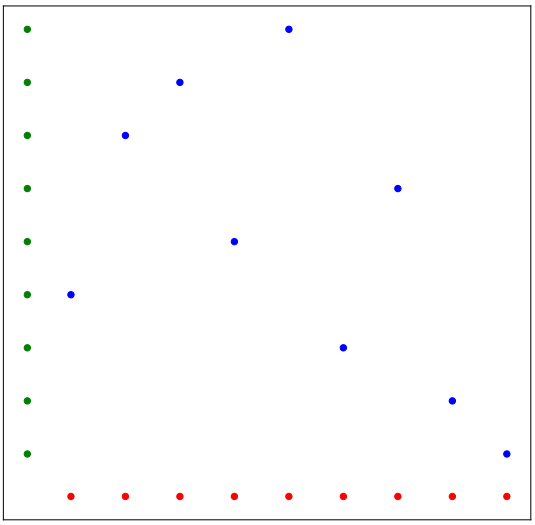


Figura 3.4: Rejilla aleatoria equidistante en dimensión 2. En azul, los puntos de la rejilla, en verde y rojo, sus proyecciones.

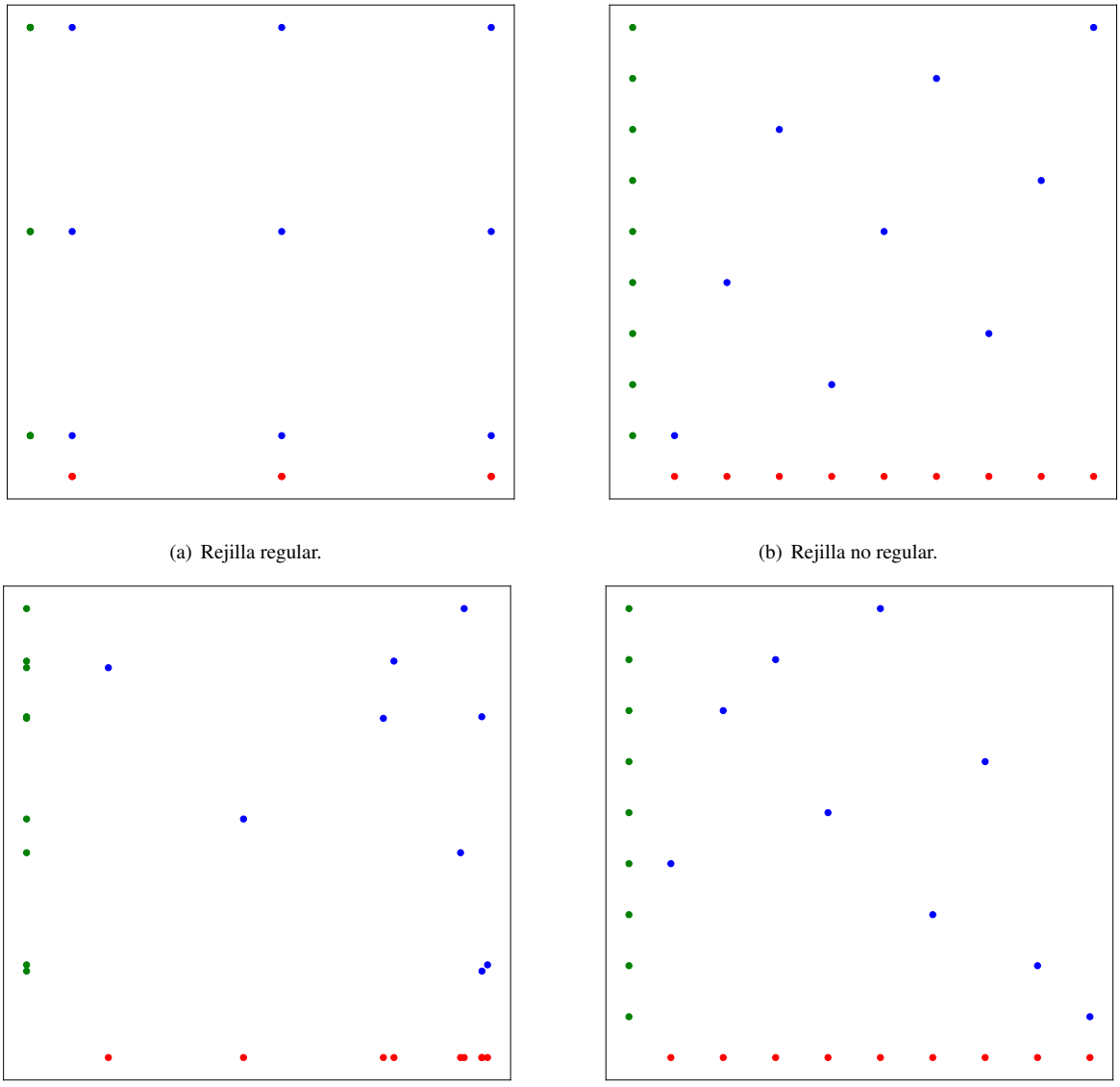
3.1.5. Comparación ilustrativa de rejillas

En esta sección vamos a comparar visualmente las rejillas de búsqueda que poseemos, tanto las originales, regular y aleatoria, como las modificaciones propuestas sobre ellas, no regular y aleatoria equidistante.

Comencemos con las construcciones de las rejillas en dimensión 2 con 9 puntos (equivalente a 3x3 puntos por dimensión), mostradas en la figura 3.4. Observamos a simple vista las diferencias más significativas entre las rejillas y sus modificaciones: la distribución de las proyecciones. Partimos de la figura 3.5(a) cuyos puntos de prueba (azules) están distribuidos aprovechando todo el cuadrado unidad, pero cuyas proyecciones sobre los ejes (puntos rojos y verdes) son escasas y además muy separadas, "desperdiciando" mucho del espacio disponible. Sin embargo, en la figura de la no regular, 3.5(b), podemos ver que los huecos en las proyecciones de la anterior rejilla se han rellenado y además de forma equidistante, optimizando al máximo el espacio disponible en cada eje con ese número de puntos (en este caso, 9).

Por otro lado, en la rejilla aleatoria, 3.5(c), observamos que los puntos están distribuidos al azar y que, por tanto, las proyecciones sobre los ejes tampoco siguen ninguna estructura, juntándose demasiado entre ellas o, por el contrario, dejando huecos significativos. Respecto a la aleatoria equidistante, 3.5(d), podemos apreciar que los puntos están seleccionados al azar pero partiendo de una rejilla regular densa, y debido a ello, sí que tenemos control sobre las proyecciones y son equidistantes entre ellas, aprovechando todo el espacio disponible en los ejes.

Observemos las rejillas en dimensión 3, con 27 puntos totales (equivalente a 3x3x3 puntos por dimensión). Vemos en la figura 3.6 que la diferencia en las proyecciones entre rejillas es aún más notable que en dimensión 2, pues ahora tenemos más combinaciones posibles que la rejilla regular no está



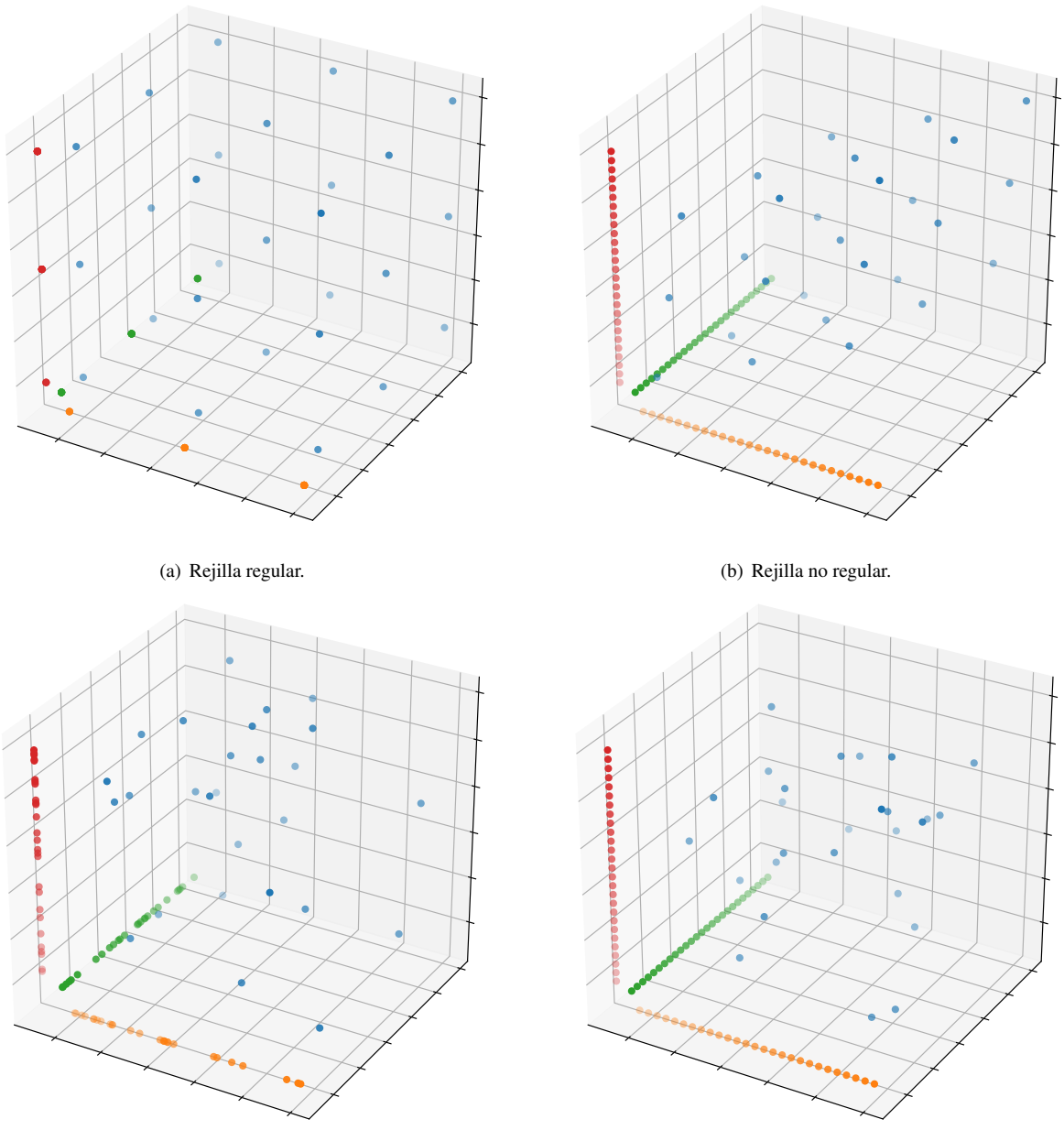
(a) Rejilla regular.

(b) Rejilla no regular.

(c) Rejilla aleatoria.

(d) Rejilla aleatoria equidistante.

Figura 3.5: Comparación de las rejillas con 9 puntos en dimensión 2. En azul, los puntos de la rejilla, en rojo y verde, sus proyecciones.



(a) Rejilla regular.

(b) Rejilla no regular.

(c) Rejilla aleatoria.

(d) Rejilla aleatoria equidistante.

Figura 3.6: Comparación de las rejillas con 27 puntos en dimensión 3. En azul, los puntos de la rejilla, en rojo, verde y naranja, sus proyecciones.

contemplando. Apreciamos, nuevamente, en la figura 3.6(a) que la distribución de los puntos (azules) ocupa uniformemente el cubo unidad, pero las proyecciones (puntos rojos, verdes y amarillos) siguen siendo escasas por cada eje (tantas como el número de puntos por dimensión) y muy separadas, por tanto. Sin embargo, la construcción de la no regular en la figura 3.6(b) deja claro el aprovechamiento de cada eje a través de las proyecciones, nuevamente equidistantes y con el número total de puntos de prueba en cada uno de ellos.

Respecto al método aleatorio, figura 3.6(c), volvemos a ver nuevamente la nube de puntos al azar con sus correspondientes proyecciones distribuidas de forma aleatoria. Por último, la rejilla aleatoria equidistante, figura 3.6(d), vuelve a mostrar puntos de forma no uniforme pero cuyas proyecciones sí se distribuyen de manera equidistante.

3.2. Detalles de implementación

En esta sección hablaremos sobre la implementación seguida para probar las rejillas creadas. Como ya comentamos, utilizaremos Python como lenguaje de desarrollo, por su rapidez y flexibilidad, y además porque cuenta con librerías adaptadas al aprendizaje automático, como es Sklearn. En concreto, usaremos la función `gridsearchCV` de esta librería, que busca los hiperparámetros del modelo utilizando validación cruzada (la estableceremos de 5 hojas) y la región de búsqueda que sea pasada por argumento.

Dado que la función tiene como entrada del espacio de búsqueda de hiperparámetros el formato de diccionario, deberemos crear una lista de diccionarios en la que cada uno de ellos deberá ser una configuración de nuestra rejilla, es decir un punto (n-dimensional) que tenga un valor de cada hiperparámetro. Por tanto, simplemente deberemos llamar a nuestras funciones que devuelven las rejillas en forma de matrices (n-dimensionales) e ir transformando cada fila de puntos en un diccionario. A este diccionario deberemos añadirle los nombres de cada hiperparámetro que acepta el estimador (`C`, `epsilon`, `gamma` ...). Además, contaremos con unos argumentos adicionales para indicar si el hiperparámetro estará en escala logarítmica o si es necesario darle el valor de entero (por ejemplo, el grado de un *kernel* polinómico).

Cabe destacar que la librería Sklearn ya proporciona métodos para la búsqueda con rejilla regular y la búsqueda aleatoria. En el caso de la regular, a la función `gridsearchCV` se le puede añadir arrays de cada valor que tomará cada hiperparámetro y ella se encarga de hacer el producto cartesiano de todas las combinaciones, obteniendo así una rejilla regular. En el caso del método aleatorio, la librería cuenta con la función `randomizedsearchCV` que es la equivalente a la anterior pero en el caso de puntos al azar. Sin embargo, utilizamos nuestras rejillas por comodidad, ya que las expresamos en un formato adecuado y homogéneo para poder pasarlas (independientemente de cuál de las 4 rejillas diferentes sea) a la función `gridsearchCV` y poder realizar los experimentos. Por tanto, queda mejor modularizado el proceso de esta forma.

Nota: los valores de rejilla regular propia son exactamente los mismos que los de la función proporcionada por Sklearn, por tanto, se puede comprobar (y lo hemos hecho) que los resultados de los experimentos coinciden.

Los códigos utilizados para la creación de las rejillas pueden ser consultados en el Apéndice A. En él, se encuentran las funciones que retornan la rejilla regular (A.1), la rejilla no regular (A.2), la rejilla aleatoria equidistante (A.3) y la rejilla aleatoria (A.4). También, se adjuntan los métodos necesarios para reescalar los puntos (A.5) y obtener las distancias de desplazamiento de puntos de cada dimensión (A.6).

4

EXPERIMENTOS

Una vez codificados los nuevos método de búsqueda, rejilla no regular y aleatoria equidistante, procedemos a realizar una serie de medidas y pruebas de interés para ver cómo se comporta la solución aportada. Éstos estudios serán: distancia entre cada par de puntos de la rejilla, búsqueda del mínimo de una función y uso de la librería Sklearn para evaluar el rendimiento como método de búsqueda de hiperparámetros en aprendizaje automático.

A lo largo de estas pruebas utilizaremos los 4 métodos diferentes, para realizar comparaciones entre ellos, centrándonos especialmente en el desempeño de nuestra rejilla no regular y nuestro método aleatorio equidistante. Los ya mencionados métodos serán:

- Rejilla no regular.
- Rejilla regular.
- Método aleatorio equidistante.
- Método aleatorio.

4.1. Distancia entre puntos

Realizar un estudio sobre las distancias entre cada par de puntos de la rejilla es interesante para tener una intuición sobre cómo están distribuidos los puntos, saber si están muy separados unos de otros, aproximar la distribución que sigue esa separación según la forma de la rejilla, etc.

Para realizar estas medidas, nos serviremos de la librería de Python SciPy, un software destinado a matemáticos, ingenieros y científicos. En concreto, de esta librería utilizaremos la función `spatial.distance.cdist`, que calcula la distancia entre cada par de puntos de una colección dada como entrada, en nuestro caso, la(s) propia(s) rejilla(s), y utilizaremos como norma la euclídea, que calcula la raíz cuadrada de la suma de las coordenadas al cuadrado de cada vector.

Las rejillas regular y no regular siempre serán las mismas, respectivamente, independientemente de las veces que las creemos. Sin embargo, debido a la aleatoriedad de los otros dos métodos, de-

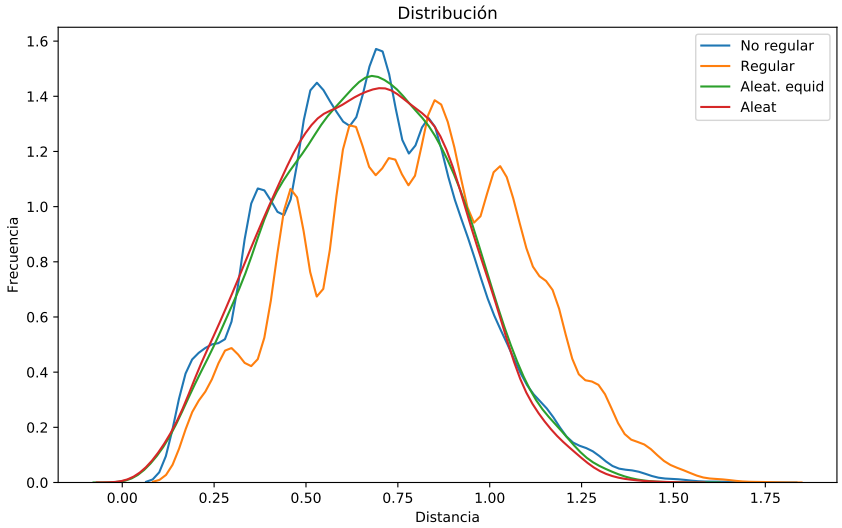
bemos realizar distintas repeticiones del proceso, para poder establecer la media de las medidas que nos interesan (la media de las medias de distancias). También obtendremos el mínimo y el máximo, quedándonos con la media de los máximos y mínimos en el caso de los métodos aleatorios. Cabe destacar que las medidas, para que sean válidas las comparaciones, serán realizadas tras escalar todas las rejillas al intervalo $[0, 1]$ en todas las dimensiones (las aleatorias ya están en dicho intervalo por construcción).

En primer lugar, vamos a establecer el número total de puntos que queremos que tengan nuestras rejillas, pongamos $6^3 = 216$. Estudiaremos cómo varían las distancias entre puntos de cada rejilla dependiendo del número de puntos que asignamos a cada dimensión. Haremos tres particiones: $[6, 6, 6]$, $[12, 6, 3]$ y $[36, 3, 2]$, es decir, de más a menos cuadrada (entendiéndolo como hipercubo en dimensiones mayores que 2) y repetiremos el experimento 100 veces para que cambien las rejillas aleatorias. Para la homogeneidad de parámetros en cada rejilla, usaremos el mismo formato indicado anteriormente, corchetes con el número de puntos que deseamos en cada dimensión. Sin embargo, las rejillas aleatorias únicamente toman el producto de número de puntos de cada dimensión para establecer el número de puntos totales (es decir, sólo les importa el número total, no el de cada coordenada).

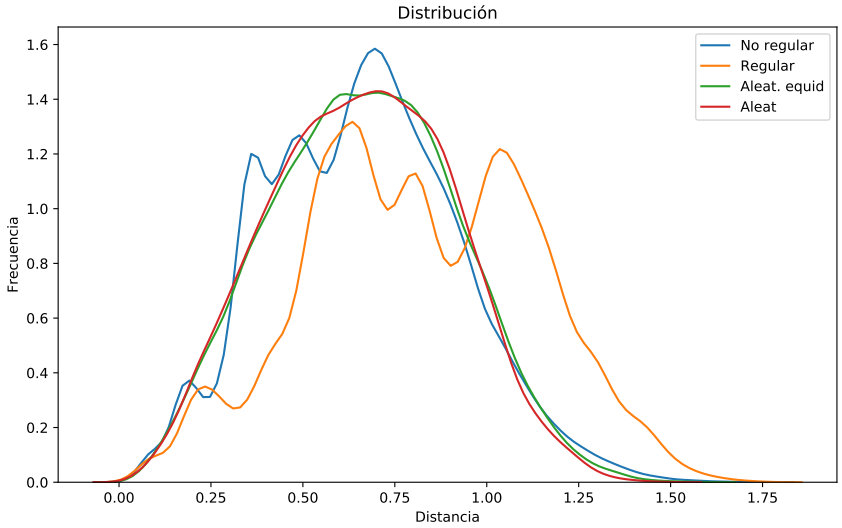
En la figura 4.1 se muestran los resultados obtenidos, en forma de histograma. Como podemos ver en ella, a medida que descompensamos el número de puntos en cada dimensión, las distancias en la rejilla regular se van separando hacia la derecha (aumenta el número de mayores distancias entre pares), mientras que los demás métodos se quedan prácticamente iguales en su distribución. Cabe señalar que la distribución que presentan estos histogramas se ha suavizado, ya que, por ejemplo, los saltos entre distancias de la rejilla regular son aún más notables de lo que se observa, pues, por construcción de la misma, hay muchos valores de separación que son iguales y hay muchos intermedios que no toma nunca. De ahí, las grandes oscilaciones que apreciamos en la gráfica regular.

Ahora, estudiamos los valores de medias, máximos y mínimos de las distancias tras el número que hayamos fijado de repeticiones. Las diferencias entre rejillas no son significativas al realizar el proceso seguido anteriormente de variar la forma de la rejilla de más a menos cuadrada. Por tanto, únicamente ilustramos las gráficas de uno de ellos, del caso $[6, 6, 6]$, pues las diferencias son sólo cuantitativas y no a nivel de proporciones entre ellas.

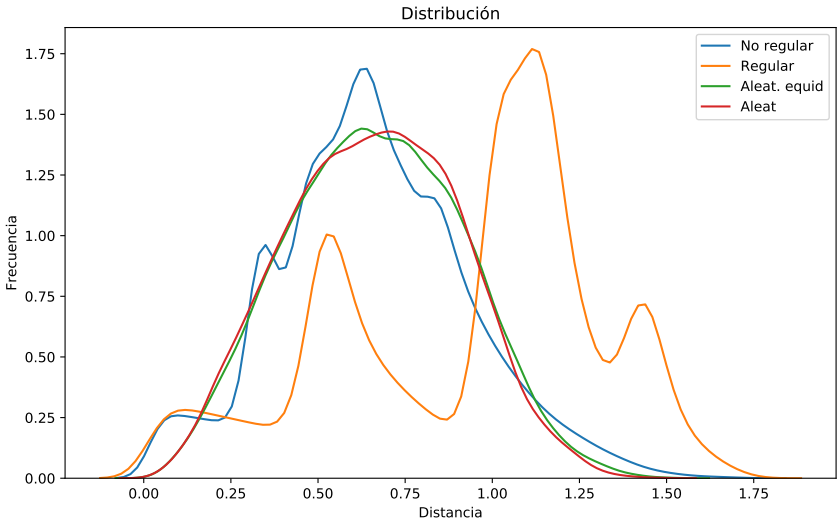
En la figura 4.2 se muestran los resultados de las medidas obtenidas. Como podríamos intuir antes de realizar las pruebas, en la figura 4.2(a) la menor media del mínimo la obtiene la rejilla aleatoria (pues no se limita cómo de cerca pueden caer dos puntos al azar, mientras que en las demás rejillas esta distancia mínima sí que está “controlada“) y el segundo puesto es para el método aleatorio equidistante, ya que la componente de aleatoriedad que tiene está restringida a ciertas combinaciones. Respecto al mayor máximo (en media), figura 4.2(b), se alcanza en las rejillas regular y no regular, ya que, por construcción, toman los valores de los extremos del hipercubo (el vector con todo $0's$ y el vector con todo $1's$). Si nos fijamos en la media, figura 4.2(c), la regular se eleva por encima de las demás, las cuales se mantienen muy ajustadas entre sí por debajo de la primera.



(a) Histograma de distancias entre puntos de partición 6,6,6.

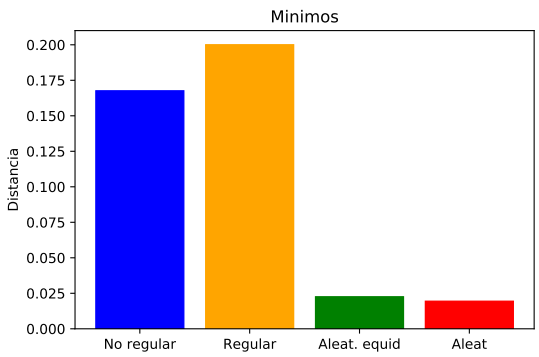


(b) Histograma de distancias entre puntos de partición 12,6,3.

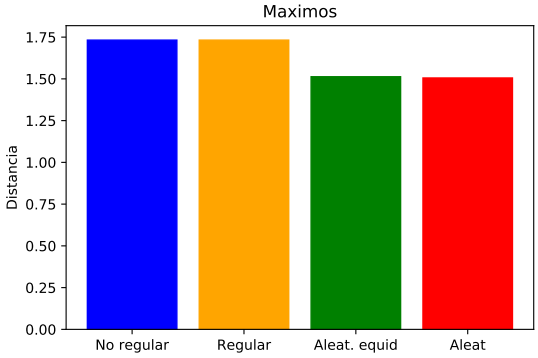


(c) Histograma de distancias entre puntos de partición 36,3,2.

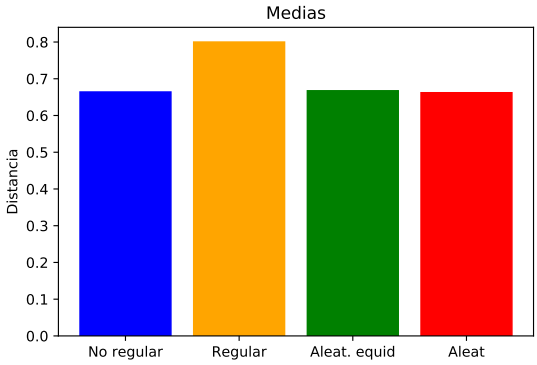
Figura 4.1: Comparación de distribuciones de distancia entre puntos.



(a) Mínimo.



(b) Máximo.



(c) Media.

Figura 4.2: Gráficos de barras de las medidas estadísticas (en media) de cada rejilla.

4.2. Mínimo de función

La siguiente parte de las pruebas consiste en un problema ya más cercano a lo que realmente estamos buscando. Vamos a, partiendo de las rejillas anteriores, buscar el mínimo valor de una función generada aleatoriamente.

La función a evaluar consistirá en un polinomio del grado que fijemos y de las dimensiones, es decir, número de variables, que elijamos. Los coeficientes de cada uno de los términos serán números aleatorios en el intervalo $[-1, 1]$ y el polinomio tendrá todas las combinaciones posibles de variables y grados escogidos. Además, para evitar que el mínimo se vaya a infinito, consideraremos el polinomio generado en valor absoluto.

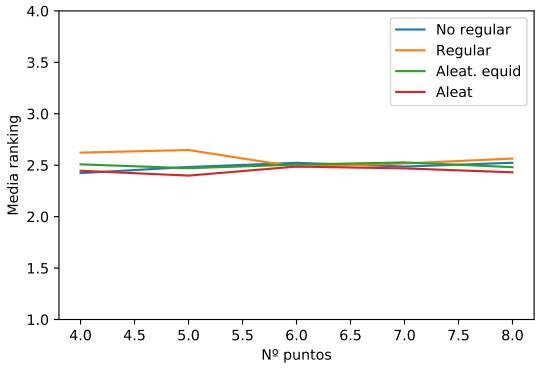
Por ejemplo, el polinomio resultante al elegir dos variables (x e y) y grado 2 sería del tipo:

$$f(x, y) = |0{,}92x^2y^2 + 0{,}78x^2y - 0{,}12x^2 + 0{,}29xy^2 - 0{,}15xy + 0{,}08x + 0{,}20y^2 + 0{,}43y + 0{,}07|$$

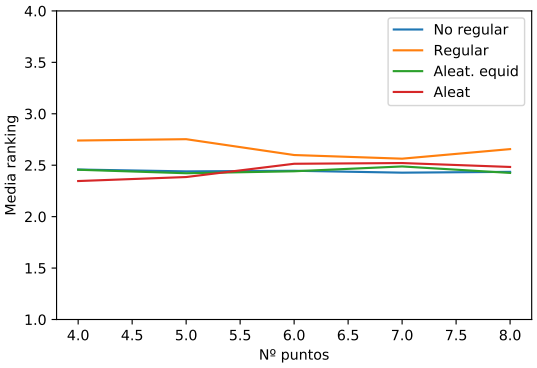
Para que se puedan evaluar tanto valores positivos como negativos, escalaremos nuestras rejillas (actualmente en $[0, 1]$) al intervalo, por ejemplo, $[-1, 1]$ en cada una de las dimensiones, de forma que nuestro espacio de pruebas será el hipercubo con extremo inferior en el vector con todo $-1's$ y con extremo superior el vector con todo $1's$.

El procedimiento a seguir es sencillo. Partimos de un número de puntos por cada dimensión. Con esto, vamos a realizar un número fijo de repeticiones en las que se van creando nuevas rejillas (recordamos que en el caso de la rejilla regular y no regular siempre serán las mismas) y se van evaluando nuevos polinomios aleatorios para encontrar el mínimo valor en los puntos de cada una de ellas. Una vez obtenidos los cuatro mínimos, elaboraremos un *ranking* por cada iteración, en el que la primera posición será para la rejilla que ha conseguido el mínimo absoluto (el más bajo) y así sucesivamente. Este *ranking* lo crearemos usando la función `scipy.stats.rankdata`. El proceso lo iremos iterando sobre el número de puntos por cada dimensión, para ver cómo evoluciona el *ranking* de mínimos dependiendo del número de posibilidades a evaluar por eje.

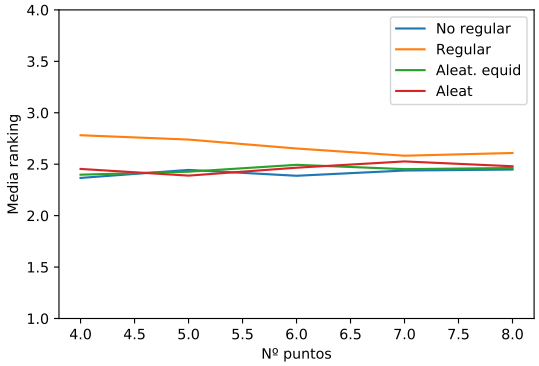
Cada prueba se ha realizado con los resultados de 1000 repeticiones. En las subfiguras de la figura 4.3 podemos ver cómo evolucionan los *rankings* a medida que incrementamos el número de puntos. Además, también se ha ido elevando el grado de los polinomios que las rejillas están minimizando. Es claro ver que a medida que aumentamos el grado, la separación entre la rejilla regular y las otras es mayor, es decir, tiene un peor rendimiento al tener una posición media de *ranking* mayor. Los otros tres métodos (no regular, aleatorio equidistante y aleatorio) se mantienen muy parejos en todas las pruebas, no observando diferencias destacables. También, esta bajada de rendimiento de la rejilla regular ocurre cuando pasamos de dimensión 3 a dimensión 4, es decir, aumentando el número de variables de los polinomios.



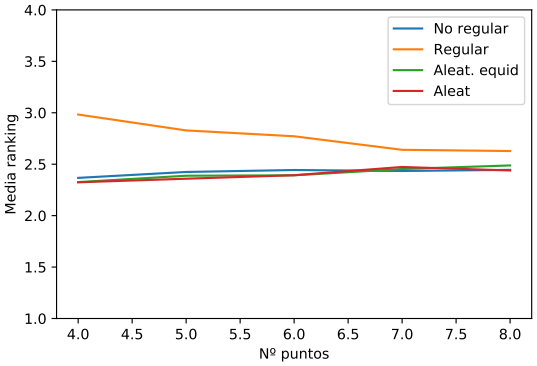
(a) Media de *ranking* en dimensión 3 y polinomio de grado 2.



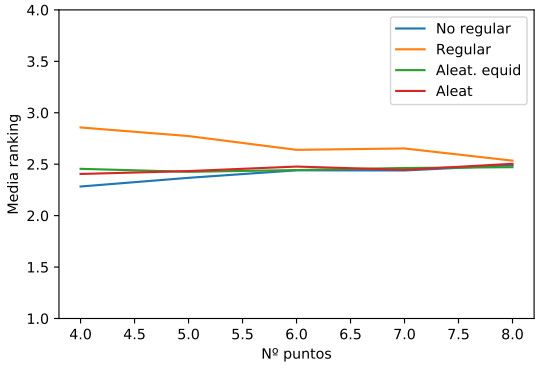
(b) Media de *ranking* en dimensión 4 y polinomio de grado 2.



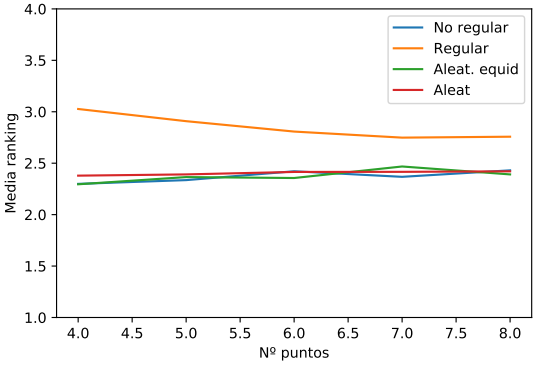
(c) Media de *ranking* en dimensión 3 y polinomio de grado 4.



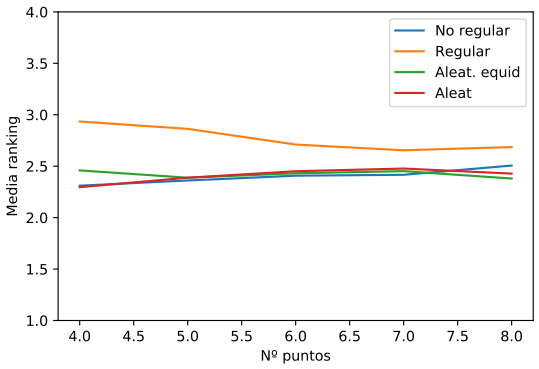
(d) Media de *ranking* en dimensión 4 y polinomio de grado 4.



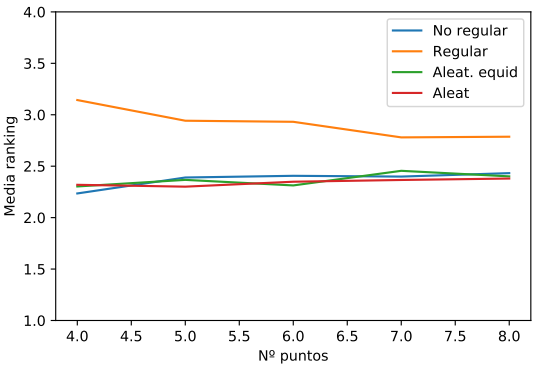
(e) Media de *ranking* en dimensión 3 y polinomio de grado 6.



(f) Media de *ranking* en dimensión 4 y polinomio de grado 6.



(g) Media de *ranking* en dimensión 3 y polinomio de grado 8.



(h) Media de *ranking* en dimensión 4 y polinomio de grado 8.

Figura 4.3: Media de *rankings* de mínimos obtenidos en dimension 3 y 4 cambiando el grado del polinomio.

Podemos usar las funciones proporcionadas por la librería *Orange* de Python, y obtener la diferencia crítica, según el Test de Bonferroni [10]. Este test se basa en la creación de un umbral en el que, por encima de él, la diferencia entre dos medias será significativa. Utilizando la función `evaluation.computeCD`, pasando como argumento las medias de *rankings* y el número de repeticiones, obtenemos un valor de distancia crítica, $D = 0{,}148$. Es decir, si las medias de dos rejillas tienen mayor diferencia que ese valor, se considerará que es significativa. Observando las gráficas de grado 6 y 8, vemos que la diferencia entre la regular y las demás es, claramente, mayor que D, y entre las otras tres es menor, pues están prácticamente pegadas. Para constatar en qué gráficas y en qué momentos se cumple la cota, podemos consultar la tabla 4.1.

Por tanto, podemos concluir que el rendimiento de la rejilla regular empeora cuanto mayor es la complejidad del polinomio a evaluar, ya que elevando el número de variables y elevando el grado de éste, obtenemos más posibles combinaciones, aprovechando mejor el espacio disponible los otros tres métodos. Además, podemos decir que el rendimiento en este problema de la rejilla no regular, aleatoria equidistante y aleatoria es estadísticamente similar.

Dimensión	Grado	Nª puntos	No Regular	Regular	Aleat equid	Aleatoria
3	2	4	2.423	2.6225	2.5085	2.446
		5	2.4825	2.6475	2.471	2.399
		6	2.523	2.484	2.506	2.487
		7	2.487	2.517	2.527	2.469
		8	2.5235	2.5645	2.481	2.431
	4	4	2.3665	2.7815	2.398	2.454
		5	2.4445	2.7395	2.427	2.389
		6	2.3875	2.6525	2.494	2.466
		7	2.4385	2.5825	2.452	2.527
		8	2.448	2.609	2.463	2.48
	6	4	2.283	2.857	2.455	2.405
		5	2.3675	2.7735	2.426	2.433
		6	2.4405	2.6395	2.443	2.477
		7	2.439	2.653	2.463	2.445
		8	2.4905	2.5345	2.471	2.504
	8	4	2.3105	2.9345	2.459	2.296
		5	2.361	2.863	2.388	2.388
		6	2.4075	2.7105	2.431	2.451
		7	2.4165	2.6545	2.452	2.477
		8	2.506	2.686	2.38	2.428
4	2	4	2.458	2.74	2.456	2.346
		5	2.44	2.753	2.422	2.385
		6	2.446	2.599	2.441	2.514
		7	2.4275	2.5635	2.488	2.521
		8	2.4355	2.6565	2.425	2.483
	4	4	2.367	2.983	2.326	2.324
		5	2.4245	2.8285	2.388	2.359
		6	2.4435	2.7715	2.392	2.393
		7	2.434	2.64	2.453	2.473
		8	2.445	2.628	2.488	2.439
	6	4	2.299	3.027	2.295	2.379
		5	2.3355	2.9085	2.365	2.391
		6	2.4215	2.8075	2.356	2.415
		7	2.3675	2.7485	2.468	2.416
		8	2.4305	2.7575	2.391	2.421
	8	4	2.235	3.143	2.303	2.319
		5	2.39	2.942	2.367	2.301
		6	2.4065	2.9315	2.313	2.349
		7	2.399	2.78	2.455	2.366
		8	2.4325	2.7865	2.401	2.38

Tabla 4.1: Tabla comparativa de *ranking* medio tras 1000 repeticiones. Aquí se puede comprobar qué medias superan la diferencia crítica $D = 0,148$ y esa diferencia es estadísticamente significativa o no.

4.3. Búsqueda de hiperparámetros en una SVR

Por último, procedemos a las pruebas principales de nuestro proyecto: la optimización de los hiperparámetros para entrenar un modelo. Para ello, utilizaremos la función `gridsearchCV` de la librería de Sklearn [1]. Esta función consta de los siguientes elementos:

- Un estimador (de regresión o de clasificación).
- Parámetros a tomar.
- Un esquema de validación cruzada.
- Una función de puntuación (*score*).

Usaremos como estimador una Máquina de Vectores Soporte (*SVM*) para regresión (*SVR; Support Vector Regression*) usando la clase `sklearn.svm.SVR` de Sklearn. Utilizaremos un *kernel* polinomial, pues es el que tiene más hiperparámetros para optimizar (hasta 5). Los parámetros serán nuestras rejillas, como explicamos anteriormente, convertidas a diccionarios. Respecto a la validación cruzada la utilizaremos de 5 hojas y la función de puntuación será la que incorpora el estimador.

Los experimentos serán realizados con el *dataset* de diabetes [11], un conjunto de datos característico de los problemas de regresión, ya integrado en la librería que utilizamos. En él, se recogen los resultados de 442 pacientes con diabetes, con 10 variables de referencia: edad, sexo, índice de masa corporal, presión arterial promedio y seis mediciones de suero sanguíneo para cada uno de los enfermos. Además, se proporciona una medida cuantitativa de la progresión de la enfermedad un año después de su comienzo (la característica objetivo, a predecir).

El significado de cada hiperparámetro en la SVM con *kernel* polinomial $(\gamma\langle x, x'\rangle + \tau)^d$ es:

- `C` : parámetro de penalización del término de error.
- `epsilon` (ε) : define un margen de tolerancia donde no se da ninguna penalización a los errores.
- `degree` (d): grado del polinomio en el *kernel* polinomial.
- `gamma` (γ) : coeficiente del *kernel*.
- `coef0` (τ) : término independiente del *kernel*.

Con esto fijado, realizaremos experimentos incrementales en el número de hiperparámetros.

1.– Experimento 1: `C`, `epsilon`.

2.– Experimento 2: `C`, `epsilon`, `degree`.

3.– Experimento 3: `C`, `epsilon`, `degree`, `gamma`.

4.– Experimento 4: `C`, `epsilon`, `degree`, `gamma`, `coef0`.

Cuando no estén presentes los hiperparámetros en el experimento, los valores por defecto serán:

- `degree` $= 5$.
- `gamma` $= \frac{1}{d}$ con d el número de características del dataset.
- `coef0` $= 1$.

Tras haber configurado los elementos descritos, realizamos el entrenamiento del modelo con las cuatro rejillas y la ya mencionada función, `gridsearchCV`. Cabe destacar que al igual que en el anterior experimento de la sección 4.2, consideramos las rejillas con el mismo número de puntos en cada parámetro. Además, vamos incrementando el número de puntos por cada hiperparámetro. Una vez fijados los datos de cada experimento, procedemos a lanzarlos.

Experimento 1

El primer experimento consistirá en ir variando los parámetros `C` y `epsilon` mediante las diferentes rejillas. Los intervalos en los que tomarán valores estos hiperparámetros son:

- `C` $\in [-3, 3]$ en escala logarítmica.
- `epsilon` $\in [-5, 0]$ en escala logarítmica.

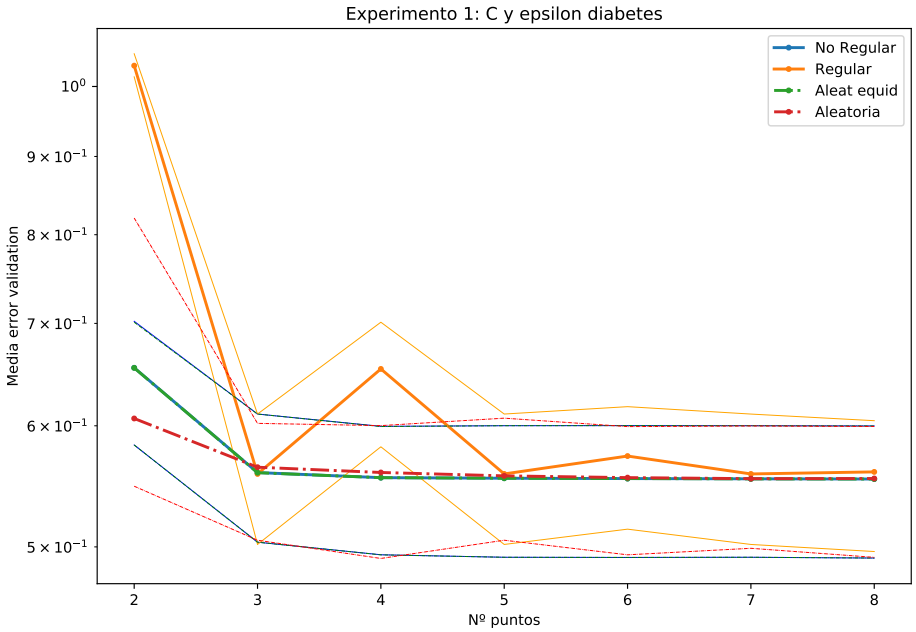


Figura 4.4: Gráfica de resultados del experimento 1. Se muestra el error medio de validación de cada rejilla, junto con el error máximo y mínimo.

Los resultados de lanzar el experimento un total de 15 repeticiones se muestran en la figura 4.4. Se representa la media del error en validación de cada rejilla (líneas más gruesas en el medio) junto con los máximos y mínimos de dicho error (líneas finas superiores e inferiores).

Podemos ver que las líneas que representan el error en la rejilla regular son menos estables, pues tienen grandes oscilaciones cuando hay pocos puntos en cada eje, obteniendo así una "banda" entre el máximo y el mínimo con un mayor error que los otros métodos. De hecho, con 2 puntos empieza con un gran error y por tanto, peor rendimiento. La rejilla no regular y la aleatoria equidistante (los dos métodos que hemos creado) coinciden en hasta 3 decimales de puntuación durante todo el experimento, por tanto, representan visualmente la mismas líneas tanto de media como de máximo y mínimo. Respecto a la rejilla aleatoria, con 2 puntos comienza con una media ligeramente inferior, y un máximo mayor y mínimo menor, pero con 3 puntos ya la diferencia entre todas sus líneas (media, máximo y mínimo) no es significativa respecto a las rejillas no regular y aleatoria equidistante.

Por tanto, en este experimento, el rendimiento de la rejilla regular con pocos puntos ($<$5) es un poco peor que el de las otras, pero al final con el incremento de puntos la diferencia se hace poco relevante. También, se aprecia que la banda de error de la aleatoria es más ancha que la banda de la no regular, lo que indica que aunque a veces la aleatoria consigue un mejor valor, cuando consigue un mal resultado es peor que el de la no regular, que parece ser más estable en ese aspecto.

Experimento 2

En el segundo experimento iremos variando los parámetros `C`, `epsilon` y `degree`, en los intervalos:

- `C` $\in [-3, 3]$ en escala logarítmica.
- `epsilon` $\in [-5, 0]$ en escala logarítmica.
- `degree` $\in [2, 10]$ entero.

Los resultados tras 15 repeticiones e incrementando el número de puntos por hiperparámetro se recogen en la figura 4.5. Podemos observar que las oscilaciones del error de la rejilla regular se han suavizado un poco, obteniendo unos valores próximos a los otros métodos a partir de 4 puntos por hiperparámetro. Al comienzo, la rejilla aleatoria equidistante tiene un máximo error bastante elevado, pero la media se mantiene por debajo de la regular y pareja a las otras dos restantes. La aleatoria prácticamente tiene el mismo rendimiento que la aleatoria equidistante a partir de 2 puntos.

Por tanto, en este experimento podemos decir que, nuevamente, el intervalo de error de la rejilla regular es superior con pocos puntos, pero que el rendimiento entre todas las rejillas se va igualando a partir de los 4 puntos por variable (esta vez con una ligera peor puntuación de la no regular).

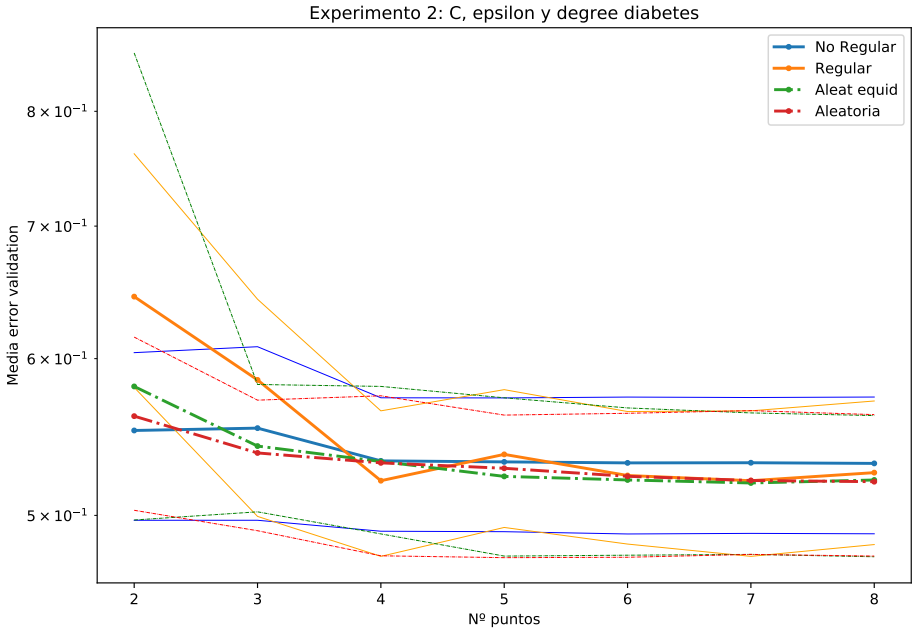


Figura 4.5: Gráfica de resultados del experimento 2. Se muestra el error medio de validación de cada rejilla, junto con el error máximo y mínimo.

Experimento 3

En este tercer experimento, volveremos a añadir otro hiperparámetro más para optimizar, por lo que iremos variando `C`, `epsilon`, `degree` y `gamma`. Los intervalos que tomarán dichas variables serán:

- `C` $\in [-3, 3]$ en escala logarítmica.
- `epsilon` $\in [-5, 0]$ en escala logarítmica.
- `degree` $\in [2, 10]$ entero.
- `gamma` $\in [log(\frac{10^{-1}}{d}), log(\frac{10^{1}}{d})]$ en escala logarítimica, con d el número de características del *dataset*. En este caso, $d = 10$, el intervalo queda $[-2, 0]$ en escala logarítmica.

Tras 10 repeticiones del proceso, obtenemos la figura 4.6. En ella vemos que las líneas de error se han suavizado respecto a los anteriores experimentos. Con 2 puntos por variable, la media de error de la aleatoria equidistante es un poco más baja que la de las demás, pero a partir de los 3 puntos por parámetro, el error de las tres medidas (media, máximo y mínimo) es muy similar en los cuatro métodos, con un ligero mayor error en media y en mínimo de la regular.

Por tanto, de este experimento concluimos que todas las rejillas tienen un rendimiento similar a partir de 4 puntos por variable. Por debajo de esa marca, la “banda“ de error de la regular es más estrecha que en las demás, lo que quiere decir que no consigue puntuaciones tan malas, pero tampoco tan buenas como los límites de las demás.

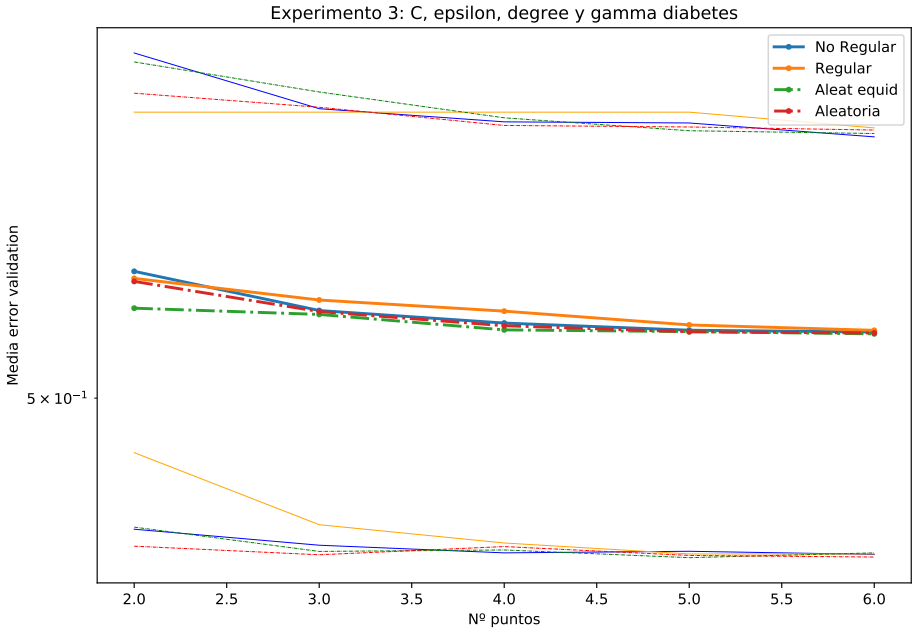


Figura 4.6: Gráfica de resultados del experimento 3. Se muestra el error medio de validación de cada rejilla, junto con el error máximo y mínimo.

Experimento 4

En el cuarto experimento, añadimos el último hiperparámetro disponible con el *kernel* polinomial, quedándonos las variables `C`, `epsilon`, `degree`, `gamma` y `coef0`. Los intervalos que tomarán serán:

- `C` $\in [-3, 3]$ en escala logarítmica.
- `epsilon` $\in [-5, 0]$ en escala logarítmica.
- `degree` $\in [2, 10]$ entero.
- `gamma` $\in [log(\frac{10^{-1}}{d}), log(\frac{10^{1}}{d})]$ en escala logarítimica, con d el número de características del *dataset*. En este caso, $d = 10$, el intervalo queda $[-2, 0]$ en escala logarítmica.
- `coef0` $\in [0, 1]$.

El proceso lo repetimos un total de 10 veces y obtenemos la figura 4.7. En ella observamos un comportamiento muy similar al del experimento anterior. A partir de 4 puntos por variable, todos los métodos confluyen, prácticamente, a una misma línea, tanto en media como en máximo y mínimo. Por debajo de los 4 puntos, vemos un error medio ligeramente menor en la regular, probablemente porque una buena puntuación sea obtenida al evaluar en los extremos de las variables (con 2 puntos).

Luego, de este último experimento concluimos lo mismo que en el anterior: el rendimiento de todas las rejillas es similar, y con pocos puntos, la "banda" de la regular es más estrecha, de tal forma que su mejor puntuación es peor que la mejor puntuación de los otros métodos.

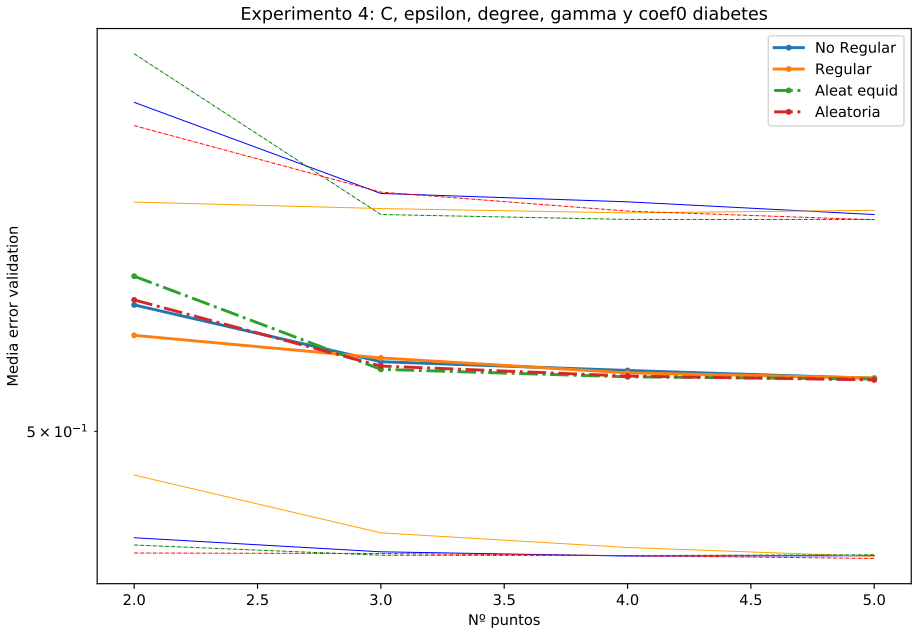


Figura 4.7: Gráfica de resultados del experimento 4. Se muestra el error medio de validación de cada rejilla, junto con el error máximo y mínimo.

5

Conclusiones y trabajo futuro

En esta sección, hablaremos sobre las conclusiones obtenidas tras todo el proceso, así como del trabajo futuro, es decir, tareas pendientes que se podrían realizar posteriormente.

5.1. Conclusiones

Nuestro objetivo era encontrar un método de optimización de hiperparámetros en el aprendizaje automático, partiendo de uno de los métodos más utilizados a día de hoy, la búsqueda en rejilla regular. Para ello, creamos dos modificaciones de esta rejilla para que las proyecciones de los puntos sean equidistantes: el método de rejilla no regular y el método aleatorio equidistante, y estudiamos su comportamiento en los experimentos.

Inicialmente, realizamos un estudio para determinar las distancias entre pares de puntos en cada una de las rejillas implementadas. De aquí, se pudo comprobar que las distancias medias eran mayores en la rejilla regular, por tanto, en general, sus puntos están más separados en media que en los otros métodos. Respecto a la máxima distancia (en media), es poco relevante, pues el mayor valor se alcanza en las rejillas no regular y regular, por construcción, ya que toman los extremos del hipercubo. Respecto a la mínima distancia (en media), lógicamente se alcanza en las aleatorias, pues no tienen una estructura definida previamente y no limitamos la distancia que puede haber entre dos puntos al azar (aunque en la aleatoria equidistante sí se restringe a ciertas combinaciones). Un factor que afecta a las distancias en la rejilla regular es la forma (simetría) de la rejilla. Cuanto menos cuadrada la hacemos, es decir, tomando muchos más puntos en unas variables que en otras, la distribución indica que aumenta el número de mayores distancias en la rejilla regular, mientras que en los otros métodos se mantiene como si de una rejilla cuadrada se tratara. Por tanto, la regular es más sensible (en el sentido de separación entre los puntos) a esta asimetría que nuestros dos métodos implementados y la rejilla aleatoria.

Después, probamos la eficiencia de los métodos en la minimización de funciones aleatorias, en este caso, polinomios. Realizamos iteraciones en el número de puntos por cada variable, también variando el grado del polinomio y el número de variables (dimensión). Para medir los resultados creamos un

ranking de posiciones según el mínimo absoluto que había alcanzado cada rejilla al evaluar en todos sus puntos, y cada proceso lo repetíamos 1000 veces para poder establecer la media de estos puestos. Las conclusiones de esta prueba fueron claras: la rejilla regular en media de *ranking* es peor que los otros métodos, que se mantienen muy parejos en todas las pruebas de esta sección. Además, a medida que aumenta la complejidad del polinomio, es decir, aumentamos el grado o aumentamos la dimensión del mismo, la media en *ranking* de la rejilla regular se hace mayor (peor posición) que los otros métodos, que siguen siendo estadísticamente similares como indica la distancía crítica del Test de Bonferroni.

Por último, nos centramos en el objetivo principal: ver el rendimiento de los métodos optimizando los hiperparámetros de un modelo de aprendizaje automático. Utilizando la librería de Sklearn y su función `gridsearchCV`, lanzamos experimentos incrementales en el número de hiperparámetros para evaluar. Con los experimentos 1 y 2, concluimos que la rejilla regular tenía un rendimiento ligeramente inferior cuando había pocos puntos por variable, pero la diferencia se iba reduciendo a medida que se aumentaba el número de puntos, hasta prácticamente igualarse en rendimiento todos los métodos. Los experimentos 3 y 4 parecen indicar que ninguna rejilla es mejor que otra, pues están muy parejas y cada vez más pegadas a medida que aumenta el número de puntos por parámetro. Se puede destacar que, en estos dos últimos experimentos, las mejores puntuaciones (las del mínimo error de validación) de la rejilla regular son peores que las mejores puntuaciones de los otros métodos, por lo que en media funciona prácticamente igual, pero su mejor desempeño no llega a ser tan bueno como en las demás.

Como conclusión final, podemos decir que las nuevas rejillas diseñadas (junto con la aleatoria) tienen menor media de distancias entre sus puntos y una mejor actuación a la hora de minimizar nuestras funciones aleatorias, respecto a la rejilla regular original. Sin embargo, la comparación en las pruebas de entrenamiento del modelo y ajustando hiperparámetros no es sencilla, y los resultados, aunque prometedores, no han sido tan concluyentes como cabría esperar. Este problema depende del conjunto de datos, los rangos de valores que toma cada hiperparámetro, si hay dimensión poco efectiva o no, etc, factores que se deberán estudiar en el futuro.

5.2. Trabajo futuro

Como hemos comentado, los experimentos de Sklearn no son sencillos de realizar, ya que hay que tener en cuenta las múltiples formas de configurar la rejilla, el conjunto de datos utilizado, el estimador, el *kernel*, etc. A esto, hay que añadir que el entrenamiento del modelo puede ser computacionalmente más lento dependiendo de lo mencionado anteriormente, es decir, el tiempo de ejecución es un factor muy importante en estas pruebas, lo que provoca que hayan quedado tareas pendientes para un futuro.

Para una mayor simplificación del problema, las rejillas las hemos considerado cuadradas para los experimentos, es decir, con el mismo número de puntos en cada hiperparámetro. En la vida real no

todos las variables cuentan con el mismo número de puntos, por tanto, se podrían realizar pruebas variando la forma de la rejilla y otorgando más puntos para evaluar en unos parámetros que en otros, para ver cómo afecta esto al entrenamiento del modelo y al posterior error de validación.

El conjunto de datos utilizado ha sido el de *diabetes*, característico de los problemas de regresión y ya integrado en la librería Sklearn. Se ha utilizado este por emplear menos tiempo de ejecución en el entrenamiento que otros probados anteriormente. Sin embargo, los hiperparámetros cobran más o menos importancia dependiendo del conjunto de datos, por tanto, para próximos experimentos se podría variar el *dataset* para ver cómo varía el rendimiento de cada una de las rejillas y ver cuántos de estos son más favorables para cada método.

Además de esto, sería conveniente realizar los experimentos con redes neuronales, pues cuentan con gran cantidad de hiperparámetros para poder realizar las configuraciones de las rejillas. Junto a esto, gracias a los estudios mencionados en el artículo de *Bergstra* [2] podríamos tener una idea de qué redes utilizar para que entre en juego la dimensión poco efectiva de una función, explicada en el capítulo 2. De esta forma, podríamos sacar provecho de la distribución de los puntos de las nuevas rejillas implementadas, potenciando el efecto de las múltiples proyecciones equidistantes con las que éstas cuentan.

Bibliografía

[1] Scikit-learn. https://scikit-learn.org//.

[2] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. 2012.

[3] John R. Koza, Forrest H. Bennett, David Andre, and Martin A. Keane. *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming*. 1996.

[4] Christopher Michael Bishop. *Pattern Recognition and Machine Learning*. 2006.

[5] Xavier Amatriain. What are hyperparameters in machine learning?, 2016. https://www.quora.com/What-are-hyperparameters-in-machine-learning.

[6] Richard O. Duda, David G. Stork, and Peter E.Hart. *Pattern Classification*. Second edition, 1973.

[7] Sunny Srinidhi. Different types of validations in machine learning, 2018. https://blog.contactsunny.com/data-science/different-types-of-validations-in-machine-learning-cross-validation.

[8] Ronald Iman. Latin hypercube sampling. January 1999.

[9] Latin hypercube sampling (lhs). https://icme.hpc.msstate.edu/mediawiki/index.php/Latin_Hypercube_Sampling_(LHS).

[10] Dr. Bill McNeese. Comparing multiple treatment means: Bonferroni's method, 2009. https://www.spcforexcel.com/knowledge/comparing-processes/bonferronis-method.

[11] Dataset loading utilities. https://scikit-learn.org/stable/datasets/index.html#diabetes-dataset.

Apéndices

A

Código de creación de las rejillas

Código A.1: Función de creación de la rejilla regular.

```
class Regular:

    def rejilla_regular(numbers):
        vectores = []
        puntos_total = np.prod(np.array(numbers))
        cont = 0
        for number in numbers:
            vector = []
            if(cont == len(numbers)-1):
                reps = 1
            else:
                reps = np.prod(numbers[cont+1:])
            bloques = puntos_total//(number*reps)
            for b in range(0, bloques):
                for i in range(0, number):
                    for r in range(0, reps):
                        vector.append(i+1)
            vectores.append(vector)
            cont = cont+1
        # reescalado dependiendo del problema
        vectores = escalar_0_1(vectores)
        return vectores
```

Código A.2: Función de creación de la rejilla no regular, a partir de la regular.

```
class No_Regular:

    def rejilla_no_regular(numbers):
        # rejilla regular sin reescalar
        vectores = rejilla_regular(numbers);
        distancias = crear_distancias(numbers);
        for v in range(0, len(vectores)):
            for element in vectores[v]:
                cont = 0
                indices = [j for j in range(len(vectores[v])) if vectores[v][j]==element]
                for i in indices:
                    vectores[v][i] = vectores[v][i] + cont*distancias[v]
                    cont=cont+1
        vectores = escalar_0_1(vectores)
        return vectores
```

Código A.3: Función de creación de la rejilla aleatoria equidistante.

```
class Aleatoria_Equidistante:

    def rejilla_aleat_equid(numbers):
        puntos_totales = np.prod(numbers)
        dim = len(numbers)
        xvals = np.linspace(0, 1, puntos_totales)
        vectores = [xvals]
        for j in range(0, dim-1):
            vals = xvals.copy()
            random.shuffle(vals)
            vectores.append(vals)
        return vectores
```

Código A.4: Función de creación de la rejilla aleatoria, con distribución uniforme en $[0, 1]$.

```
class Aleatoria:

    def rejilla_aleat(numbers):
        total_puntos = np.prod(numbers)
        dim = len(numbers)
        # distribución uniforme en [0,1]
        vectores = np.random.rand(total_puntos,dim)
        # queremos los ejes como filas
        vectores = vectores.transpose()
        return vectores
```

Código A.5: Función de reescalado de puntos al intervalo $[0, 1]$.

```
def escalar_0_1(matriz):
    nueva_matriz = []
    for vector in matriz:
        vector = np.array(vector)
        if(max(vector) == min(vector)):
            vector = np.zeros(len(vector))
        else:
            vector = (vector-min(vector))/(max(vector)-min(vector))
        nueva_matriz.append(vector)
    return nueva_matriz
```

Código A.6: Función que devuelve las distancias a desplazar en cada dimensión.

```
def crear_distancias(numbers):
    distancias = []
    for i in range(0, len(numbers)):
        d_i = 1/np.prod(np.array(numbers[:i]+numbers[i+1:]))
        distancias.append(d_i)

    return distancias
```